

# Checkpointing and Rollback-Recovery for Distributed Systems*

Richard Koo**
Sam Toueg***

TR 85-706
October 1985

TECHNICAL REPORT



Department of Computer Science
Cornell University
Ithaca, New York

12

# Checkpointing and Rollback-Recovery for Distributed Systems*

Richard Koo**
Sam Toueg***

TR 85-706
October 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853



* The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

** This author was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grants DCR-8412582 and MCS 83-03135.

*** This author was supported by the National Science Foundation under grant MCS 83-03135.

# Checkpointing and Rollback-Recovery for Distributed Systems*

*Richard Koo*†

*Sam Toueg*‡

Computer Science Department
Cornell University
Ithaca, New York 14853

## ABSTRACT

We consider the problem of bringing a distributed system to a consistent state after transient failures. We address the two components of this problem by describing a distributed algorithm to create consistent checkpoints, as well as a rollback-recovery algorithm to recover the system to a consistent state. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints. Similarly, when a process restarts after a failure, a minimal number of additional processes are forced to restart with it. Our algorithms require each process to store at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

**Keywords:** fault-tolerance, checkpoint, rollback-recovery, distributed systems, consistent state.



*The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defence position, policy, or decision.

†This author was supported by the Defence Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grants DCR-8412582 and MCS 83-03135.

‡This author was supported by the National Science Foundation under grant MCS 83-03135.

## 1. Introduction

Checkpointing and rollback-recovery are well-known techniques that allow processes to make progress in spite of failures [Rand78]. The failures under consideration are transient problems such as hardware errors and transaction aborts, i.e., those that are unlikely to recur when a process restarts. With this scheme, a process takes a checkpoint from time to time by saving its state on stable storage [Lamp79]. When a failure occurs, the process rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution.

We first identify consistency problems that arise in applying this technique to a distributed system. We then propose a checkpoint algorithm and a rollback-recovery algorithm to restart the system from a consistent state when failures occur. Our algorithms prevent the well-known "domino effect" as well as liveness problems associated with rollback-recovery. In contrast to previous algorithms, they are fault-tolerant and involve a minimal number of processes. With our approach, each process stores at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

The paper is organized as follows: We discuss the notion of consistency in a distributed system in section 2 and describe our system model in section 3. In section 4 we identify the problems to be solved. Sections 5 and 6 contain the checkpoint and rollback-recovery algorithms respectively. The algorithms are extended for concurrent executions in section 7. In section 8 we consider optimizations. Sections 9 and 10 contain a discussion and our conclusion.

## 2. Consistent Global States in Distributed Systems

The notion of a consistent global state is central to reasoning about correctness in distributed systems. It was initially studied in [Rand75, Russ77, Pres83] and later formalized by Chandy and Lamport [Chan85]. We summarise the ideas in [Chan85]:

In a distributed computation, an *event* at a process $p$ can be a spontaneous change of $p$'s state, or the sending or receipt of a message by $p$. Event $a$ *directly happens before* event $b$ if and only if

(1) there exist states $s_1$, $s_2$, and $s_3$ such that event $a$ changes $p$'s process state from $s_1$ to $s_2$ and event $b$ changes $p$'s process state from $s_2$ to $s_3$; or

(2) event $a$ is the sending of a message $m$ by a process $p$ and event $b$ is the receiving of $m$ by another process $q$.

The transitive closure of the *directly happens before* relation is the *happens before* relation. If event $a$ happens before event $b$, $b$ *happens after* $a$. (We abbreviate *happens before*, "before" and *happens after*, "after".)

The *local state* of a process at time 0 is its initial state; the *local state* of a process at time $t$ is the state resulting from applying the sequence of events occurring in $(0, t]$ to its initial state. If a process has failed by time $t$, its local state at $t$ is undefined. A *global state* of a system at time $t$ is the set of all processes' local states at $t$. The *state of a channel* at time $t$ is the set of messages sent over that channel but not yet received at $t$. We can depict the occurrences of events over time with a time diagram, in which horizontal lines are time axes of processes, points are events, and arrows represent messages from the sending process to the receiving process. In this representation, a global state is a cut dividing the time diagram into two halves. The channel states are the arrows (messages) that cross the cut. Figure 1 is a time diagram for a system of four processes.

Informally, a cut (global state) in the time diagram is *consistent* if no arrow starts on the right hand side and ends on the left hand side of it. This notion of consistency fits the observation that a message cannot be received before it is sent in any temporal frame of reference. For example, the cuts $c$ and $c'$ in Figure 1 are consistent and inconsistent cuts, respectively. The channel states corresponding to cut $c$ consists of one message in the channel from $p$ to $q$, and one in the channel from $s$ to $r$. Readers are referred to [Chan85] for a formal discussion of consistent global states.

## 3. System Model

The distributed system considered in this paper has the following characteristics:

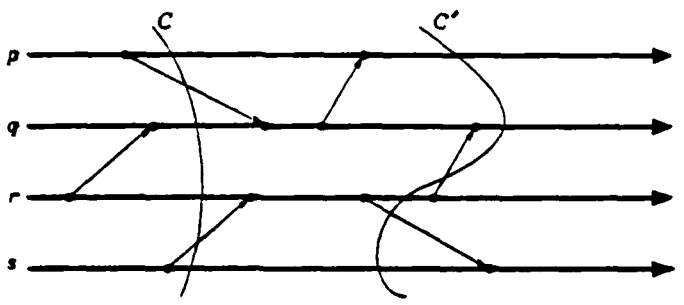


FIG. 1. *Consistent and inconsistent cuts in a distributed system.*

(1) Processes do not share memory or clocks. They communicate via messages sent through reliable first-in-first-out (FIFO) channels with variable non-zero transmission time.

(2) Processes fail by stopping, and whenever a process fails, all other processes are informed of the failure in finite time. We assume that processes' failures never partition the communication network.

We want to develop our algorithms under the weakest possible set of assumptions. In particular, we do not assume that the underlying system is a database transaction system ([Fisc82] and [Jose85]). This special case admits simpler solutions: the mechanisms that ensure atomicity of transactions can hide inconsistencies introduced by the failure of a transaction. Furthermore, we do not assume that processes are deterministic: this simplifying assumption is made in previous results (e.g., [Stro85] and [Jose85]).

## 4. Identification of Problems

A checkpoint is a saved state of a process. A set of checkpoints, one per process in the system, is consistent if the saved states form a consistent global state. For example, consider the system history in Figure 2. Process $p$ takes a checkpoint at time $X$ and sends a message to $q$ some time later. After receiving this message, $q$ takes a checkpoint at time $Y$. Subsequently, $p$ fails and restarts from the checkpoint taken at $X$. The global state at $p$'s restart is inconsistent because $p$'s local state shows that no message has been sent to $q$, while $q$'s local state shows that a message from $p$ has been received. If $p$ and $q$ are processes supervising a customer's accounts at different banks, and the message transfers funds from $p$ to $q$, the customer will have the funds at *both* banks when $p$ restarts. This inconsistency persists even if $q$ is forced to roll back and restart from its checkpoint taken at $Y$.

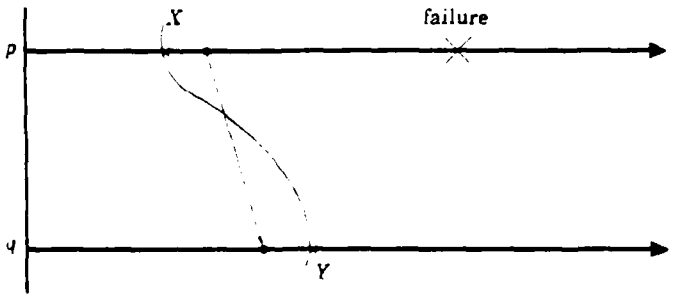


FIG. 2. *Inconsistent checkpoints.*

The problem of ensuring that the system recovers to a consistent state after transient failures has two components: checkpoint creation and rollback-recovery; we examine each one in turn.

## 4.1. Checkpoint Creation

There are two approaches to creating checkpoints. With the first approach, processes take checkpoints independently and save all checkpoints on stable storage. Upon a failure, processes must find and agree upon a consistent set of checkpoints among the saved ones. The system is then rolled back to and restarted from this set of checkpoints [Ande79, Russ80, Wood81, Hadz82].

With the second approach, processes coordinate their checkpointing actions such that each process saves only its most recent checkpoint, and the set of checkpoints in the system is guaranteed to be consistent. When a failure occurs, the system restarts from these checkpoints [Tami84].

A disadvantage of the first approach has long been recognized [Rand75, Pres83] and is named the "domino effect". We illustrate this effect in Figure 3. In this example, processes $p$ and $q$ have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for $p$ and $q$, except the initial one at $\{X_0, Y_0\}$. Consequently, after $p$ fails, both $p$ and $q$ must roll back to the starting point of the computation. For time-critical applications that require a guaranteed rate of progress, such as real time process control, this behavior results in unacceptable delays. An additional disadvantage of independent checkpoints is the large amount of stable storage required for the saved states.

To avoid these drawbacks, we pursue the second approach. In contrast to [Tami84], our method ensures that when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints.

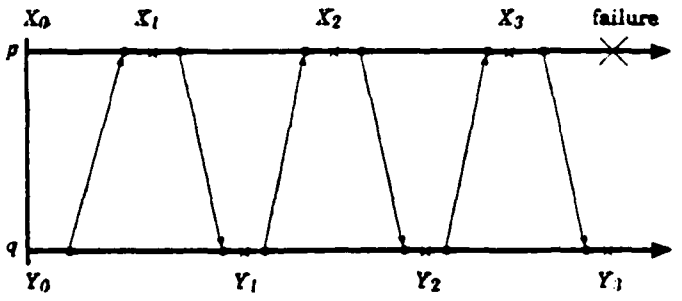


FIG. 3. *"Domino effect" following a failure.*

## 4.2. Rollback-Recovery

Rollback-recovery from a consistent set of checkpoints appears deceptively simple. The following scheme seems to work: Whenever a process rolls back to its checkpoint, it notifies all other processes to also roll back to their respective checkpoints. It then installs its checkpointed state and resumes execution. Unfortunately, this simple recovery method has a major flaw. In the absence of synchronization, processes cannot all recover (from their respective checkpoints) simultaneously. Recovering processes at different times introduces a liveness problem as illustrated below.

Consider two processes $p$ and $q$. Figure 4 illustrates their histories up to the time $p$ fails. Process $p$ fails before receiving the message $n_1$, rolls back to its checkpoint, and notifies $q$. Then $p$ recovers, it sends $m_2$ and receives $n_1$. After $p$'s recovery, $p$ has no record of sending $m_1$, while $q$ has a record of its receipt. Therefore, the global state is inconsistent. To restore consistency, $q$ must also roll back (to "forget" the receipt of $m_1$), and notify $p$. Note that after $q$ rolls back, $q$ has no record of sending $n_1$ while $p$ has a record of its receipt. Hence, the global state is inconsistent again, and upon notification of $q$'s rollback, $p$ must roll back a second time. After $q$ recovers, $q$ sends $n_2$ and receives $m_2$. Suppose $p$ rolls back before receiving $n_2$ as shown in Figure 5. With the second rollback of $p$, the sending of $m_2$ is "forgotten". To restore consistency, $q$ must roll back a second time. After $p$ recovers it receives $n_2$, and upon notification of $q$'s rollback, it must roll back a third time. It is now clear that $p$ and $q$ can be forced to roll back forever, even though no additional failures occur.

Our rollback-recovery algorithm solves this liveness problem. It tolerates failures that occur during its execution, and forces a minimal number of processes to roll back after a failure. In [Tami84], a single failure forces the system to roll

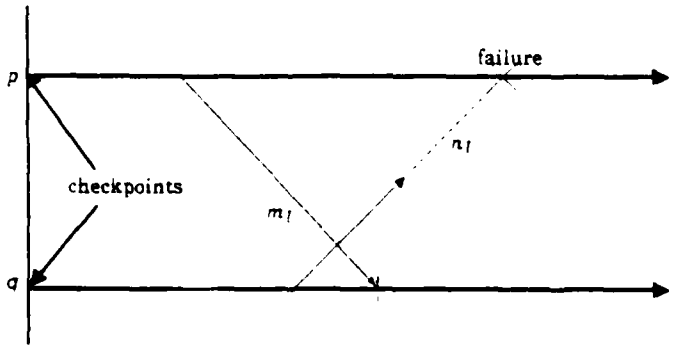


FIG. 4. *Histories of p and q up to p's failure.*

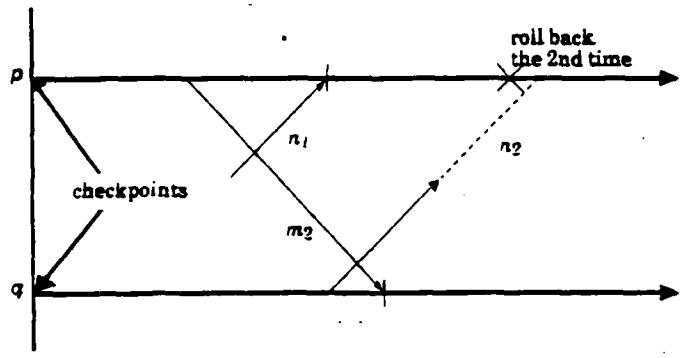


FIG. 5. *History of p and q up to p's 2nd rollback.*

back as a whole. Furthermore, the system crashes (and does not recover) if a failure occurs while it is rolling back.

## 5. Checkpoint Creation

### 5.1. Naive Algorithms

From Figure 2 it is obvious that if every process takes a checkpoint after every sending of a message, and these two actions are done atomically, the set of the most recent checkpoints is always consistent. But creating a checkpoint after every send is expensive. We may naively reduce the cost of the above method with a strategy such as "every process takes a checkpoint after every $k$ sends, $k>1$" or "every process takes a checkpoint on the hour". However, the former can be shown to suffer domino effects by a construction similar to the one in Figure 3, while the latter is meaningless for a system that lacks perfectly synchronized clocks.

### 5.2. Classes of Checkpoints

Our algorithm saves two kinds of checkpoints on stable storage: permanent and tentative. A permanent checkpoint cannot be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint. When the context is clear, we call permanent checkpoints "checkpoints".

Consider a system with a consistent set of permanent checkpoints. A checkpoint algorithm is *resilient* to failures if the set of permanent checkpoints in the system is still consistent after the algorithm terminates, even if some processes fail during its execution. Consider systems where processes cannot afford to take a checkpoint after every send, or systems where processes cannot combine the sending of a message and the taking of a checkpoint atomically. For these systems, checkpoint algorithms must store at least two checkpoints in stable storage in

order to be resilient to failures.

**Theorem 1:** No resilient checkpoint algorithms exist that take only permanent checkpoints.

*Proof:* By contradiction. Suppose such an algorithm A exists. Consider the following scenario: $p$ and $q$ are processes in the system. Suppose that by time $t$, $t>0$, $p$ has received a message $m_q$ from $q$, and $q$ a message $m_p$ from $p$. At time $t$, process $p$ decides to use A to take a checkpoint. Let A finish by time $t'$, and suppose process $p$ takes a permanent checkpoint $C_{p,t_p}$ at time $t_p$, such that $t<t_p<t'$. Since the set of checkpoints at the termination of A must be consistent, process $q$ must also have taken a permanent checkpoint $C_{q,t_q}$ at time $t_q$, such that $t<t_q<t'$. Let $d$ be the minimum time required for the failure of a process to be detected. Depending on whether $t_p \leq t_q$ or $t_p > t_q$, we construct another run of A in which one process fails, to show that A is not resilient.

Case 1: $t_p \leq t_q$. Let $q$ fail in the time interval $(\max(t, t_q-d), t_q)$. Process $p$ discovers the failure after $t_q$, hence after $t_p$. (See Figure 6.) Consequently, $C_{p,t_p}$ is taken although $C_{q,t_q}$ is not. Since $C_{p,t_p}$ is a permanent checkpoint that cannot be undone, and $q$ fails before making a permanent checkpoint, the sending of $m_q$ is "forgotten" forever while the receipt of $m_q$ is always "remembered", no matter what A does after $p$ detects the failure. Hence, contrary to our assumption, Algorithm A is not resilient.

Case 2: $t_p > t_q$. Let $p$ fail in the time interval $(\max(t, t_p-d), t_p)$. The

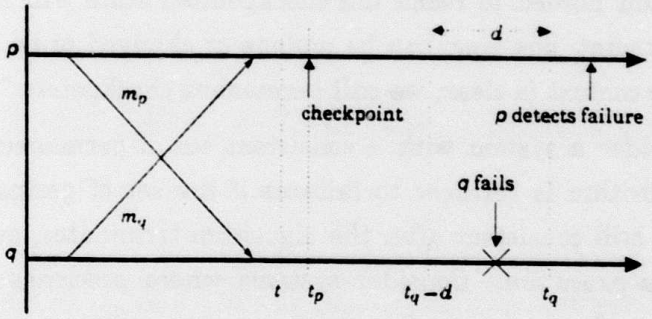


FIG. 6. *The scenario when* $t_p \leq t_q$ and $q$ fails.

rest of the proof is analogous to Case 1. □

Theorem 1 shows that besides the impractical "naive" algorithm described in section 5.1, any resilient checkpoint algorithm must store at least two checkpoints on stable storage.

## 5.3. Our Checkpoint Algorithm

We assume that the algorithm is invoked by a single process that wants to take a permanent checkpoint. We also assume that no failures occur in the system. In section 5.3.4 we extend the algorithm to handle failures, and in section 7 we describe concurrent invocations of this algorithm.

### 5.3.1. Motivation

To create consistent checkpoints, processes can execute an algorithm that is patterned on two-phase-commit protocols. In the first phase, the initiator $q$ takes a tentative checkpoint and requests all processes to take tentative checkpoints. If $q$ learns that all processes have taken tentative checkpoints, $q$ decides all tentative checkpoints should be made permanent; otherwise, $q$ decides tentative checkpoints should be discarded. In the second phase, $q$'s decision is propagated and carried out by all processes. Since all or none of the processes take permanent checkpoints, the most recent set of checkpoints is always consistent.

However, our goal is to force a minimal number of processes to take checkpoints. The above algorithm is modified as follows: A process $p$ takes a tentative checkpoint after it receives a checkpoint request from $q$ *only if* $q$'s tentative checkpoint records the receipt of a message from $p$, while $p$'s latest permanent checkpoint does not record the sending of that message. Process $p$ determines whether this condition is true using the label appended to $q$'s request. This labeling scheme is described below.

Messages that are not sent by the checkpoint or rollback-recovery algorithms are *system* messages. Every system message $m$ contains a label $m.l$. Each process appends outgoing system messages with monotonically increasing labels. We define $\perp$ and $\top$ to be the smallest and largest labels, respectively. For any processes $r$ and $p$, let $m$ be the last message that $r$ received from $p$ after $r$ took its last permanent or tentative checkpoint. Define:

$$last\_rmsg_r(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Also, let $m$ be the first message that $r$ sent to process $p$ after $r$ took its last

permanent or tentative checkpoint. Define:

$$first\_smsg_r(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise} \end{cases}.$$

When $q$ requests $p$ to take a tentative checkpoint, it appends $last\_rmsg_q(p)$ to its request; $p$ takes the checkpoint if $\perp < first\_smsg_p(q) \leq last\_rmsg_q(p)$.

### 5.3.2. Description

Process $p$ is a *ckpt_cohort* of $q$ if $q$ has taken a tentative checkpoint, and $last\_rmsg_q(p) > \perp$ before the tentative checkpoint was taken. The set of ckpt_cohorts of $q$ is denoted $ckpt\_cohort_q$. Every process $p$ keeps a variable $willing\_to\_ckpt_p$ to denote its willingness to take checkpoints. Whenever $p$ cannot be interrupted to run the checkpoint algorithm, $willing\_to\_ckpt_p$ is "no". The initiator $q$ starts the checkpoint algorithm by making a tentative checkpoint and sending a request "take a tentative checkpoint and $last\_rmsg_q(p)$" to all $p \in ckpt\_cohort_q$. A process $p$ *inherits* this request if $willing\_to\_ckpt_p$ is "yes" and $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$. After $p$ inherits a request, it takes a tentative checkpoint and sends "take a tentative checkpoint and $last\_rmsg_p(r)$" requests to all $r \in ckpt\_cohort_p$. If $p$ receives but does not inherit a request from $q$, $p$ replies $willing\_to\_ckpt_p$ to $q$.

After $p$ sends out its requests, it waits for replies that can be either "yes" or "no", indicating a ckpt_cohort's acceptance or rejection of $p$'s request. If at least one reply is "no", $willing\_to\_ckpt_p$ becomes "no"; otherwise $willing\_to\_ckpt_p$ is unchanged. Process $p$ then sends $willing\_to\_ckpt_p$ to the process whose request $p$ has inherited.

If all the replies from its ckpt_cohorts arrive and are all "yes", the initiator decides to take all tentative checkpoints permanent. Otherwise the decision is to undo all tentative checkpoints. This decision is propagated in the same fashion as the request "take a tentative checkpoint" was delivered. Between the times a process $p$ takes a tentative checkpoint and it receives the decision from the initiator, $p$ does not send any system messages. Also, after processes take new permanent checkpoints, they may discard their previous checkpoints.

The algorithm is presented in Figure 7. For simplicity, we create a fictitious process called *daemon* to assume the initiation and decision tasks of the initiator. In practice, daemon is a part of the initiator process.

---

[1] **await** does not prevent a process from receiving messages.

Daemon process:

```
send(initiator, "take a tentative checkpoint and T");
await(initiator, reply);[1]
if reply = "yes" then
  send(initiator, "take tentative checkpoint permanent")
else
  send(initiator, "undo tentative checkpoint")
fi.
```

All processes $p$:

```
INITIAL STATE:
  first_smsg_p(daemon) = T;
  willing_to_ckpt_p = { "yes"  if p is willing to take a checkpoint
                      { "no"   otherwise                          ;

UPON RECEIPT OF "take a tentative checkpoint and last_rmsg_q(p)" from q DO
  if willing_to_ckpt_p and last_rmsg_q(p) ≥ first_smsg_p(q) > ⊥ then
    take a tentative checkpoint;
    for all v∈ckpt_cohort_p, send(v, "take a tentative checkpoint and last_rmsg_p(v)");
    for all v∈ckpt_cohort_p, await(v, willing_to_ckpt_v);
    if ∃ v∈ckpt_cohort_p, willing_to_ckpt_v = "no" then willing_to_ckpt_p ← "no" fi;
  fi;
  send(q, willing_to_ckpt_p);
od

UPON FIRST RECEIPT OF m="take tentative checkpoint permanent" or
  m="undo tentative checkpoint" DO
  if m="take tentative checkpoint permanent" then
    take tentative checkpoint permanent;
  else
    undo tentative checkpoint;
  fi;
  for all v∈ckpt_cohort_p, send(v, m);
od.
```

FIG. 7. *Algorithm C1: the Checkpoint Algorithm*

### 5.3.3. Proof of Correctness

We consider a single invocation of the algorithm, and we assume no process fails in the system.

**Lemma 2.** Every process inherits a request to take a tentative checkpoint at most once.

*Proof*: Immediately after a process $p$ inherits a request it takes a tentative checkpoint. From the time $p$ takes this checkpoint to the time it receives the initiator's decision, $p$ does not send any system messages. Therefore, during this interval of time, $first\_smsg_p(q) = \perp$ for all $q$. Process $p$ does not inherit additional requests during the execution of the algorithm. □

**Lemma 3:** Every process terminates its execution of Algorithm C1.

*Proof*: Any process that executes C1 without making a tentative checkpoint clearly terminates. Let $p$ be a process that takes a tentative checkpoint. By lemma 2, $p$ inherits a request to take a tentative checkpoint at most once. Consequently, to prove that C1 terminates at $p$, it suffices to prove that after $p$ takes a tentative checkpoint, it does not wait forever for either the "yes" or "no" from its ckpt_cohorts, or the initiator's decision.

Let $q$ be a ckpt_cohort of $p$. If $q$ inherits $p$'s request to take a tentative checkpoint, it sends $willing\_to\_ckpt_q$ to $p$ before it waits for the initiator's decision. If $q$, on the other hand, does not inherit $p$'s request, it sends $p$ $willing\_to\_ckpt_q$ immediately after receiving $p$'s request. Therefore, there can be no deadlock involving $p$ waiting for replies from its ckpt_cohorts and a ckpt_cohort of $p$ waiting for the initiator's decision.

Suppose that $p$ is in a deadlock waiting for replies from its ckpt_cohorts. Then there exists a circular chain of processes $p = p_0$, $\cdots$, $p_k$ $(k \geq 1)$ such that for $0 \leq i \leq k$, $p_i$ waits forever for its ckpt_cohort, $p_{i+1 \bmod k}$, to send $willing\_to\_ckpt_{p_{i+1 \bmod k}}$. If $p_i$ waits forever for $p_{i+1 \bmod k}$, $p_{i+1 \bmod k}$ must have inherited a request from $p_i$. Since the initiator does not inherit any requests, it is not in the chain. And since there is only one initiator, there must exist a process $q$ such that for some $i$, $p_i$ inherits a request from $q$, and $q \neq p_i$ for all $i$. But $p_i$

contradicts lemma 2 by inheriting two requests: one from $q$ and one from $p_{i-1 \bmod k}$. Consequently, no deadlock can exist and $p$ will receive replies from all its ckpt_cohorts.

Since every process receives replies from all its ckpt_cohorts, the initiator will receive replies from all its ckpt_cohorts to decide on the tentative checkpoints. Its decision is guaranteed to reach all processes that have taken tentative checkpoints because all processes will pass on the decision and messages are always delivered. Thus we have shown that no process waits forever for replies from its ckpt_cohorts or the initiator's decision. □

The next lemma shows that C1 takes a consistent set of checkpoints.

**Lemma 4:** If the set of checkpoints in the system is consistent before the execution of Algorithm C1, the set of checkpoints in the system is consistent after the termination of C1.

*Proof:* Without loss of generality, assume new checkpoints are taken in C1. The proof is by contradiction. Suppose the set of checkpoints after C1 terminates is not consistent. Then there must exist two processes $p$ and $q$ such that $p$ sent $q$ a message $m$ after making its permanent checkpoint, and $q$ received $m$ before making its permanent checkpoint. Since all checkpoints are consistent before the execution of C1, $q$ must have taken its permanent checkpoint during this execution. Before $q$ took a tentative checkpoint in C1, $last\_rmsg_q(p) \geq m.l$; therefore, $p$ was in $ckpt\_cohort_q$ and received a request to take a tentative checkpoint from $q$. When $p$ received the request, $willing\_to\_ckpt_p$ had to be "yes" because $q$ cannot have taken its tentative checkpoint permanent otherwise. Moreover, if $p$ had not taken a tentative checkpoint when $q$'s request arrived, $last\_rmsg_q(p) \geq first\_smsg_p(q)$ because $first\_smsg_p(q) \leq m.l$. Hence, process $p$ took a tentative checkpoint after sending $m$. Process $p$, however, must take its tentative checkpoint permanent if $q$ takes its permanent. Consequently $p$ takes a permanent checkpoint after sending $m$, a contradiction. □

We now show that the number of processes that take new permanent checkpoints during the execution of Algorithm C1 is minimal. Let $P=\{p_0, p_1, \cdots, p_k\}$ be the set of processes that take new permanent checkpoints in C1, where $p_0$ is the initiator of C1. Let $C(P)=\{c(p_0), c(p_1), \cdots, c(p_k)\}$ be the permanent checkpoints

taken by processes in $P$. Define an alternate set of checkpoints: $C'(P)=\{c'(p_0), c'(p_1), \cdots, c'(p_k)\}$ where $c'(p_0) = c(p_0)$ and for $1 \le i \le k$, $c'(p_i)=$ either $c(p_i)$ or the checkpoint $p_i$ had before executing C1.

**Theorem 5**: $C'(P)$ is consistent if and only if $C'(P)=C(P)$.

*Proof*: Without loss of generality, assume $|P| \ge 2$. The *if* part is by lemma 4. We show the *only if* part by contradiction. Suppose $C'(P) \neq C(P)$ and $C'(P)$ is consistent. Then there exists a nonempty subset $Q$ of $P$ such that for all process $q$ in $Q$, $c'(q) \neq c(q)$. For any processes $p$ and $q$, if $p$ inherits a checkpoint request from $q$, $q$'s tentative checkpoint is taken before $p$'s. Therefore, the inherit relation is non-circular. Because of this non-circularity and the fact that the initiator is in $Q$ (since $c'(p_0)=c(p_0)$), there exists $p_i \in Q$ such that $p_i$ inherits a checkpoint request from another process $p_j \notin Q$. Since $p_i \in P$ implies $p_j \in P$, we know that $c'(p_j) = c(p_j)$.

When $p_i$ inherits $p_j$'s request, $last\_rmsg_{p_j}(p_i) \ge first\_smsg_{p_i}(p_j) > \perp$. There exists a message $m$ such that $last\_rmsg_{p_j}(p_i)=m.l$. In $C'(P)$, the sending of $m$ is not recorded in $c'(p_i)$ since $m.l \ge first\_smsg_{p_i}(p_j)$, but the receipt of $m$ is recorded in $c'(p_j)$. Contrary to the assumption, $C'(P)$ is not consistent. □

Theorem 5 shows that if $p_0$ takes a checkpoint, then all processes in $P$ must take a checkpoint to ensure global consistency.

### 5.3.4. Coping with Failures

We now extend Algorithm C1 to handle processes' failures. We first consider the effects of failures on non-faulty processes. When failures occur, a non-faulty process may receive zero or more of the following messages:

(1) "yes" or "no" from ckpt_cohorts,

(2) "take tentative checkpoint permanent" or "undo tentative checkpoint" from the initiator.

Suppose process $p$ fails before replying "yes" or "no" to process $q$'s request. By the assumptions of section 3, $q$ will know of $p$'s failure. Process $q$ can then assume that $p$ is unwilling to take a permanent checkpoint. This assumption is correct even if $p$ has taken a tentative checkpoint before it fails, as long as $p$ undoes its tentative checkpoint when it recovers (see section 5.5). Therefore, to take care of the case of a missing "yes" or "no", it suffices to change the line in C1 from

**if** $\exists\ v \in ckpt\_cohort_p$, $willing\_to\_ckpt_v$ = "no" **then** $willing\_to\_ckpt_p \leftarrow$ "no" **fi**

to

**if** $\exists\ v \in ckpt\_cohort_p$, $willing\_to\_ckpt_v$ = "no" or $v$ has failed **then**
$willing\_to\_ckpt_p \leftarrow$ "no" **fi**.

Suppose that a process $p$ does not receive the decision regarding its tentative checkpoint. If $p$ undoes its tentative checkpoint or takes it permanent, it risks contradicting the initiator. A common practice in this situation is to have $p$ blocked until it discovers the initiator's decision [Skee82]. We will discuss ways to obviate blocking in section 8.

We now consider the recovery of faulty processes. When a process restarts after a failure, its latest checkpoint on stable storage may be tentative or permanent. If this checkpoint is tentative, the recovering process must decide whether to discard it or to take it permanent. The decision is made as follows:

Suppose the recovering process is the initiator. The initiator knows that every process that has taken a tentative checkpoint is still blocked waiting for its decision. Hence it is safe for the initiator to decide to undo the tentative checkpoints and send this decision to its ckpt_cohorts.

If the recovering process is not the initiator, it must discover the initiator's decision regarding tentative checkpoints. It may contact either the initiator or those processes of which it is a ckpt_cohort; it follows the decision accordingly.

Now the recovering process is left with one permanent checkpoint on stable storage. Recovery is complete when it uses the rollback-recovery algorithm to be presented in section 6 to restart from this checkpoint.

Let C2 be the Algorithm C1 as modified above. C2 terminates if all processes that fail during the execution of C2 recover. At termination, the set of checkpoints in the system is consistent, and the number of processes that took new permanent checkpoints is minimal. The proofs for these properties are similar to those of C1 and are omitted.

## 6. Rollback-Recovery

We assume that the algorithm is invoked by a single process that wants to roll back and recover (henceforth denoted *restart*). We also assume that the checkpoint algorithm and the rollback-recovery algorithm are not invoked concurrently. Concurrent invocations of the algorithms are described in section 7.

## 6.1. Motivation

The rollback-recovery algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator $q$ requests all processes to indicate their willingness to restart from their checkpoints. Process $q$ decides to restart all the processes if and only if they are all willing to restart. In the second phase, $q$'s decision is propagated and carried out by all processes. We will prove that the two-phase structure of this algorithm prevents the liveness problem discussed in section 4.2. Since all or none of the processes restart, when the rollback-recovery algorithm terminates the global state is consistent.

However, our goal is an algorithm that rolls back a minimal number of processes in order to recover from a failure. If a process $p$ rolls back to a state saved before an event $e$ occurred, we say that $e$ is *undone* by $p$. With our algorithm, process $p$ must restart *only if* $q$'s rollback will undo the sending of a message to $p$. Process $p$ determines if it must restart using the label appended to $q$'s request.

For any processes $r$ and $p$, let $m$ be the last message that $r$ sent to $p$ before $r$ took its latest permanent checkpoint. Define

$$last\_smsg_r(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \top & \text{otherwise} \end{cases}.$$

When $q$ requests $p$ to restart, it appends $last\_smsg_q(p)$ to its request. Process $p$ restarts from its permanent checkpoint if $last\_rmsg_p(q) > last\_smsg_q(p)$.

## 6.2. Description

Process $p$ is a $roll-cohort$ of $q$ if $q$ can send messages to it. The set of roll-cohorts of $q$ is $roll-cohort_q$[2]. Every process $p$ keeps a variable $willing\_to\_roll_p$ to denote its willingness to roll back. The initiator $q$ starts the rollback-recovery algorithm by sending a request "prepare to roll back and $last\_smsg_q(p)$" to all $p \in roll-cohort_q$. A process $p$ *inherits* this request if $willing\_to\_roll_p$ is "yes", $last\_rmsg_p(q) > last\_smsg_q(p)$, and $p$ has not already inherited another request to roll back. After $p$ inherits the request, it sends "prepare to roll back and $last\_smsg_p(r)$" to all $r \in roll-cohort_p$; otherwise, it replies $willing\_to\_roll_p$ to $q$.

---

[2]The relationship between $roll-cohort$ and $ckpt\_cohort$ is not symmetric. If $p$ is a $ckpt\_cohort$ of $q$, $last\_rmsg_q(p) > \perp$ and $q$ must then be a $roll-cohort$ of $p$. On the other hand, it is possible that $p \notin ckpt\_cohort_q$ but $q \in roll-cohort_p$ because $p$ can but does not send messages to $q$.

After $p$ sends out its requests, it waits for replies from each process in $roll-cohort_p$. The reply can be an explicit "yes" or "no" message, or an implicit "no" when $p$ discovers that $r$ has failed. If at least one reply is "no", $willing\_to\_roll_p$ becomes "no", otherwise $willing\_to\_roll_p$ is unchanged. Process $p$ then sends $willing\_to\_roll_p$ to the process whose request $p$ inherits. Between the times $p$ inherits the rollback request and it receives the decision from the initiator, it does not send any system messages.

If all the replies from its roll-cohorts arrive and are all "yes", the initiator decides the rollbacks will proceed, otherwise it decides no process will roll back. This decision is propagated to all processes in the same fashion as the request "prepare to roll back" is delivered. Process $p$ blocks waiting for the discovery of the initiator's decision, if failures prevent the decision from reaching $p$. We discuss non-blocking algorithms in section 8.

The rollback-recovery algorithm is presented in Figure 8. Like the presentation of Algorithm C1, we introduce a fictitious process called *daemon* to perform functions that are unique to the initiator of the algorithm.

## 6.3. Proof of Correctness

We first assume that the rollback-recovery algorithm is invoked by a single process that wants to restart. The variable $ready\_to\_roll_p$ ensures that a process $p$ inherits at most one request to roll back. Therefore, to prove the termination of Algorithm R, it suffices to show that Algorithm R is free of deadlock and it rolls each process back at most once.

**Lemma 6:** Algorithm R is deadlock free.

*Proof:* Similar to the proof of lemma 3. □

**Lemma 7:** Every process in the system rolls back at most once.

*Proof:* Without loss of generality, assume that the initiator decides to roll back. The initiator receives replies from all its roll-cohorts only after all processes have received replies from all their respective roll-cohorts. Therefore, should a process $p$ receive a rollback request from another process $q$ after $p$ has received the initiator decision, the initiator must have decided to roll back before it received all the replies from its roll-cohorts, an impossibility. □

We next show that for each send event that is undone in Algorithm R, its corresponding receive event is also undone.

Daemon process:

```
send(initiator, "prepare to roll back and ⊥");
await(initiator, reply);
if reply = "yes" then
  send(initiator, "roll back")
else
  send(initiator, "do not roll back")
fi.
```

All processes $p$:

```
INITIAL STATE:
  ready_to_roll_p = true;
  last_rmsg_p(daemon) = ⊤;
  willing_to_roll_p = { "yes"  if p is willing to roll back
                      { "no"   otherwise                    ;

UPON RECEIPT OF "prepare to roll back and last_smsg_q(p)" from q DO
  if willing_to_roll_p and last_rmsg_p(q) > last_smsg_q(p) and ready_to_roll_p then
    ready_to_roll_p ← false;
    for all r∈roll−cohort_p, send(r, "prepare to roll back and last_smsg_p(r)");
    for all r∈roll−cohort_p, await(r, willing_to_roll_r);
    if ∃ r∈roll_cohort_p, willing_to_roll_r = "no" or r has failed
      then willing_to_roll_p ← "no" fi;
  fi;
  send(q, willing_to_roll_p);
od;

UPON RECEIPT OF m = "roll back" or
  m = "do not roll back" and ready_to_roll_p = true DO
  if m = "roll back" then
    roll back to p's permanent checkpoint;
  else
    resume normal execution;
  fi;
  for all r∈roll−cohort_p, send(r, m);
od;
```

FIG. 8. *Algorithm R: the Rollback Algorithm*

**Lemma 8:** After every process has terminated its execution of Algorithm R, for each send event that was undone, its corresponding receive event was also undone.

*Proof:* Without loss of generality, assume that the initiator decides to roll back. The proof is by contradiction. Suppose that after Algorithm R terminates, there exists a message $m$ such that the receiver $p$ did not undo the receipt of $m$ while the sender $q$ undid the sending of $m$. First, we show that $p$ inherited a request to roll back. Since $q$ cannot send system messages after inheriting a rollback request, $q$ must have sent $m$ before inheriting the request. And since $q$ undid the sending of $m$, $m.l > last\_smsg_q(p)$. Therefore, when $p$ receives $q$'s request, $last\_rmsg_p(q) \geq m.l > last\_smsg_q(p)$. In addition, the variable $willing\_to\_roll_p$ must have been "yes"; otherwise the initiator cannot have decided to roll back. Consequently, when $q$'s request reached $p$, either $p$ had already inherited a rollback request or it inherited $q$'s request.

Next we show that $p$ undid the receipt of $m$. Since $p$ and $q$ received the same decision, $p$ rolled back. There are two cases to consider:

Case 1: $m$ reached $p$ after $p$ inherited a rollback request. Obvious.

Case 2: $m$ reached $p$ before $p$ inherited a rollback request. The receipt of $m$ was not undone only if after receiving $m$ and before inheriting a rollback request, $p$ took a permanent checkpoint. However, if $p$ took a permanent checkpoint after receiving $m$ while $q$ did not take a permanent checkpoint after sending $m$ (since $q$ can undo the sending of $m$), lemma 4 will be contradicted.

In all cases, $p$ undoes the receipt of $m$ when it rolls back, contradicting our assumption. □

Lastly, we show that a minimal number of processes roll back in Algorithm R. Let $P$ be the set of processes in the system that roll back.

**Theorem 9:** After Algorithm R terminates, for each send event that is undone, its corresponding receive event is undone if and only if for all nonempty $Q \subset P$ such that $Q$ does not contain the initiator, all processes in $Q$ roll back.

*Proof:* Without loss of generality, assume $|P| \geq 2$. The *if* part is by lemma 8. We show the *only if* part by contradiction. Suppose that there exists a $Q$ such that even if all processes in $Q$ do not roll back, for each send event that is undone by Algorithm R, its corresponding receive event is undone. For any processes $p$ and $q$, if $p$ inherits a rollback request from $q$, $ready\_to\_roll_q$ becomes true before $ready\_to\_roll_p$ becomes true. Therefore, the inherit relation is non-circular. Because of this non-circularity and the fact that the initiator is in Q, there exists $q \in Q$ such that $q$ inherits a rollback request from another process $p$ outside of $Q$. Since $q \in P$, $p \in P$. When $q$ inherits $p$'s request, $last\_rmsg_q(p) > last\_smsg_p(q)$. Let $m$ be the message such that $m.l = last\_rmsg_q(p)$. If processes in $Q$ do not roll back while those in $P - Q$ do, $p$ undoes the sending of $m$ while $q$ does not undo the receipt of $m$, a contradiction. □

## 7. Interference

In this section, we consider concurrent invocations of the checkpoint and rollback-recovery algorithms. An execution of these algorithms by process $p$ is *interfered* with if any of the following events occur:

(1) Process $p$ receives a rollback request from another process $q$ while executing the checkpoint algorithm.

(2) Process $p$ receives a checkpoint request from $q$ while executing the rollback-recovery algorithm.

(3) Process $p$, while executing the checkpoint algorithm for initiator $i$, receives a checkpoint request from $q$, but $q$'s request originates from a different initiator than $i$.

(4) Process $p$, while executing the rollback-recovery algorithm for initiator $i$, receives a rollback request from $q$, but $q$'s request originates from a different initiator than $i$.

One single rule handles the four cases of interference: once $p$ starts the execution of a checkpoint [rollback] algorithm, $p$ is unwilling to take a tentative checkpoint [roll back] for another initiator, or to roll back [take a tentative checkpoint]. As a result, in all four cases, $p$ replies "no" to $q$. We can show that this rule suffices to guarantee that all previous lemmas and theorems hold despite concurrent invocations of the algorithms. This rule can, however, be modified to permit more concurrency in the system. The modification is that in case (1), instead of

sending "no" to $q$, $p$ can begin executing the rollback-recovery algorithm after it finishes the checkpoint algorithm. We cannot, however, apply a similar modification in case (2) lest deadlock may occur.

## 8. Optimization

When the initiator of the checkpoint or of the rollback-recovery algorithm fails before propagating its decision to its cohorts, it is desirable for processes not to block waiting for its recovery. To prevent processes from blocking, we can modify our algorithms by replacing the underlying two-phase commit protocol with a non-blocking three-phase commit protocol [Skee82]. However, non-blocking protocols are inherently more expensive than blocking ones [Dwor83].

We now address the following problem: after a ckpt_cohort $q$ of a process $p$ fails, $p$ is unable to take a permanent checkpoint until $q$ recovers ($p$ cannot know if the latest checkpoint of $q$ records the sendings of all messages it received from $q$). To avoid waiting for $q$'s recovery, $p$ can remove $q$ from $ckpt\_cohort_p$ by restarting from its checkpoint (using the rollback-recovery algorithm). Thereafter, process $p$ can take checkpoints.

## 9. Message Loss

Rollback-recovery can cause message loss as illustrated in Figure 9. When $p$ is rolled back to $X$ following a failure at $F$, the global state is consistent, but the message $m$ from $q$ is lost. It is lost because the set of checkpoints $\{X, Y\}$ corresponds to a consistent global state with $m$ in the channel.

One method to circumvent message loss requires that processes use transmission protocols that transform lossy channels to virtual error-free channels, e.g., sliding window protocols [Tane81]. Another method is to ensure that the most recent set of checkpoints corresponds to a consistent global state with no messages

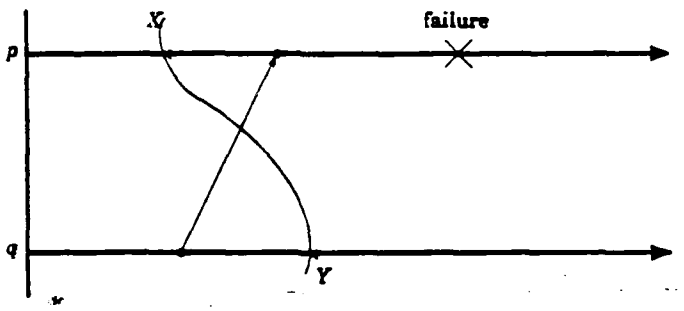


FIG. 9. *Message loss following p's rollback to X.*

in the channels. We can modify the checkpoint and rollback-recovery algorithms to satisfy this condition, but this modification increases the number of processes that are forced to take checkpoints and roll back.

## 10. Conclusion

We have presented a checkpoint algorithm and a rollback-recovery algorithm to solve the problem of bringing a distributed system to a consistent state after transient failures. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints. Similarly, when a process restarts after a failure, a minimal number of additional processes are forced to restart with it. We also show that the stable storage requirement of our algorithms is minimal.

**Acknowledgements** We would like to thank Amr El Abbadi, Ken Birman, Rance Cleaveland, and Jennifer Widom for commenting on earlier drafts of this paper.

## Bibliography


Ande79 T. Anderson, P. A. Lee and S. K. Shrivastava, System fault tolerance, in *Computing System Reliability*, T. Anderson, B. Randell (eds.) Cambridge University Press, Cambridge, 1979, pp. 153-210.

Bria84 D. Briatico, A. Ciuffoletti, and L. Simoncini, A distributed domino-effect free recovery algorithm, *Proc. of the 4th Symposium on Reliability in Distributed Software and Database Systems*, October 1984.

Chan85 K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, February 1985.

Dwor84 C. Dwork and D. Skeen, The inherent cost of nonblocking commitment, *Proc. ACM Symposium on Principles of Database Systems*, March 1983.

Fisc82 M. Fischer, N. Griffeth, and N. Lynch, Global states of a distributed system, *IEEE Transaction on Software Engineering*, May 1982, pp. 198-202

Hadz82 V. Hadzilacos, An algorithm for minimizing rollback cost, *Proc. ACM Symposium on Principles of Database Systems*, March 1982.

Jose85 T. Joseph and K. Birman, Low cost management of replicated data in fault-tolerant distributed systems, To appear in *TOCS*.

Lamp78 L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

Lamp79 B. Lampson and H. Sturgis, Crash recovery in a distributed storage system, *Xerox PARC Tech. Rep.*, Xerox Palo Alto Research Center, April 1979.

Pres83 D. L. Presotto, Publishing: A reliable broadcast communication mechanism, *Tech. Rep. UCB/CSD 83-165, Computer Science Division, University of California, Berkeley*, December 1983.

Rand75 B. Randell, System structure for software fault tolerance, *IEEE Transactions On Software Engineering*, vol. SE-1, no.2, June 1975, pp. 226-232.

Rand78 B. Randell, P.A. Lee, and P.C. Treleaven, Reliability issues in computing system design,*ACM Computing Surveys*, vol. 10, no. 2, June 1978, pp. 123-166.

Russ77 D. L. Russell, Process backup in producer-consumer systems, *Proc. ACM Symposium on Operating Systems Principles*, November, 1977.

Russ80 D. L. Russell, State restoration in systems of communicating processes, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, March 1980, pp. 183-194.

Skee82 D. M. Skeen, Crash recovery in a distributed database system, *PhD. dissertation, Computer Science Division, University of California, Berkeley*, 1982.

Stro85 R. Strom and S. Yemini, Optimistic recovery in distributed systems, *Transactions on Computer Systems*, August 1985, pp. 204-226.

Tami84 Y. Tamir and C. H. Sequin, Error recovery in multicomputers using global checkpoints, *Proc. of 13th International Conference on Parallel Processing*, August 1984.

Tane81 A. S. Tanenbaum, *Computer Networks*, Prentice Hill, New Jersey, 1981, pp. 148-164.

Wood81 W. G. Wood, A decentralized recovery control protocol, *Proc. of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.